Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 107109, 9 pages
http://dx.doi.org/10.1155/2014/107109

Research Article

Efficient Big Integer Multiplication and Squaring Algorithms for Cryptographic Applications

Shahram Jahani, Azman Samsudin, and Kumbakonam Govindarajan Subramanian

School of Computer Sciences, Universiti Sains Malaysia, Penang 11800, Malaysia

Correspondence should be addressed to Azman Samsudin; azman@cs.usm.my

Received 15 November 2013; Revised 4 July 2014; Accepted 5 July 2014; Published 24 July 2014

Academic Editor: Jin L. Kuang

Public-key cryptosystems are broadly employed to provide security for digital information. Improving the efficiency of public-key cryptosystem through speeding up calculation and using fewer resources are among the main goals of cryptography research. In this paper, we introduce new symbols extracted from binary representation of integers called Big-ones. We present a modified version of the classical multiplication and squaring algorithms based on the Big-ones to improve the efficiency of big integer multiplication and squaring in number theory based cryptosystems. Compared to the adopted classical and Karatsuba multiplication algorithms for squaring, the proposed squaring algorithm is 2 to 3.7 and 7.9 to 2.5 times faster for squaring 32-bit and 8-Kbit numbers, respectively. The proposed multiplication algorithm is also 2.3 to 3.9 and 7 to 2.4 times faster for multiplying 32-bit and 8-Kbit numbers, respectively. The number theory based cryptosystems, which are operating in the range of 1-Kbit to 4-Kbit integers, are directly benefited from the proposed method since multiplication and squaring are the main operations in most of these systems.

1. Introduction

The growth of digital technologies has an exponential trend and as a consequence the need of information security also increases even more than before [1, 2]. Cryptography is an essential tool in providing a reasonable solution for this necessity. The modern field of cryptography consists of two main areas, the symmetric-key cryptography and the public-key cryptography. The same key is used in symmetric-key cryptosystems to encrypt and decrypt a message, while the public-key cryptosystems use two keys in their protocols. Most of the public-key cryptosystems [3] use modular exponentiation in their calculation. For example, Diffie and Hellman introduced the first key exchange scheme in 1967 that is based on the modular exponentiation [4]. Few years later in 1978, one of the most used public-key cryptosystems, RSA [3], is also based on the modular exponentiation. ElGamal key exchange [5] is another example of public key that has been developed based on the modular exponentiation.

Modular exponentiation, $b = a^x \bmod n$, is a one-way function because the inverse of a modular exponentiation ($x = d \log_a b$) is a known hard problem [6–8]. To achieve a comfortable level of security, the length of the key material for these cryptosystems must be larger than 1024 bits [9], and in the near future, it is predicted that 2048-bit and 4096-bit systems will become standard [10].

Calculating modular exponentiation for a large exponent and large modulo is a costly operation and therefore improving its efficiency has become an important research issue for researchers in cryptography and mathematics. There are two main approaches currently being employed in order to improve the efficiency of modular exponentiation: improving the involved operations, exponentiation, and division, separately, and improving both of the operations simultaneously. Residue number system (RNS) [11] and Montgomery modular multiplication [12] are examples of the first approach, while binary and m-ary exponentiation or Barrett reduction [13] are instances from the second approach. This paper focuses on the second approach, by proposing a new number representation, which will improve the squaring and multiplication operations, two of the three main operations in calculating modular exponentiation [14].

Input: positive integers a and $x > 1$.
Output: $b = a^x$

(1) Set $b \leftarrow 1$ and $s \leftarrow a$.
(2) While $x \neq 0$ do the following:
  (2.1) If x is odd then $b \leftarrow b \times s$. // $(k/2)$ multiplication if $k = \log_2 b$.
  (2.2) Set $x \leftarrow \lfloor x/2 \rfloor$.
  (2.3) If $x \neq 0$ then $s \leftarrow s \times s$. // $(k-1)$ *multiplication.*
(3) Return b.

Algorithm 1: Right-to-left binary exponentiation, $\mathbf{b} = \mathbf{a}^{\mathbf{x}}$.

Input: positive integers $A = (a_n, \ldots, a_0)_r$ having $n+1$ base r digits and $B = (b_m, \ldots, b_0)_r$ having $m+1$ base r digits.
Output: the product $A \cdot B = (c_{m+n+1}, \ldots, c_0)_r$ in base r.
Note: $(uv)_r$ are two single-precision digits in base r, indicating the result of the addition.

(1) For i from 0 up to $m+n+1$ do: $c_i \leftarrow 0$.
(2) For i from 0 up to m do the following:
  (2.1) $carry \leftarrow 0$.
  (2.2) For j from 0 up to n do the following:
    (2.2.1) Compute $(uv)_r = c_{i+j} + a_j \cdot b_i + carry$, set $c_{i+j} \leftarrow v$, and $carry \leftarrow u$. // *u and v are*
  (2.3) $c_{i+n+1} \leftarrow u$. // *single-precision*
(3) Return $(c_{m+n+1}, \ldots, c_0)$. // *digits in base r*

Algorithm 2: Multiple-precision classical multiplication, CM(A, B).

The naive approach of calculating the exponentiation is by doing repetitive multiplication, which is not an efficient way for calculating large exponent. A better alternative for calculating exponentiation is by employing binary exponentiation; that is, if $b = \sum_{i=0}^{k-1}(b_i 2^i)$, where $b_i = \{0, 1\}$, then $a^b = \prod_{i=0}^{k-1}(a^{2^i b_i})$. The term a^{2^i} can be obtained by squaring the $(i-1)$th term, $a^{2^{i-1}}$. The number of operations for calculating a^b by using the naïve method is $(b-1)$ multiplications. On the other hand, the binary method requires only $(k-1)$ squarings and $k/2$ multiplications (on average), where $k = \log_2 b$ (see Algorithm 1). Consequently, improving the multiplication and squaring operations (as found in algorithm such as the right-to-left algorithm and its variants [6–8]) will inherently improve the efficiency of the exponentiation calculation [7].

2. Multiplication and Squaring Algorithms

The most well-known algorithms for multiplication of two large integers or two polynomials are classical [15], Karatsuba-Ofman's [16], Toom-Cook's [17, 18], and fast Fourier transform (FFT) multiplication algorithms [19]. In spite of all the differences in these methods, which sometimes make them apparently unrelated to each other, these methods have been founded based on the same idea, that is, how to represent a polynomial to behave efficiently in calculations. The classical method uses coefficient representation, while the other three methods use point-value representation. This representation conversion enables us to reduce the cost of convolution from $O(n^2)$ of classical method to a lower cost for point-to-point multiplication. The process of finding point-value representation from its coefficient representation is called "*evaluation*" or "*point evaluation*" and the reverse process is known as "*interpolation*." Table 1 summarizes the differences among the multiplication algorithms by their complexity, technique, and representation used.

Algorithms such as FFT and Toom-Cook have lower algorithm complexity. However, because of the preprocessing overheads such as the divide and conquer, evaluation, and interpolation, the operating cost of these algorithms is actually much higher, making them useful only when the integers are extremely large. Consequently, only classical and Karatsuba multiplication algorithms and their combination are being used in current cryptosystem. This is especially true after considering circumstances such as memory constraints and the practical finite field size.

2.1. Classical Multiplication and Squaring Algorithms. In positional numeral system [15], the natural way of multiplying numbers, known as classical multiplication algorithm, is by multiplying each digit of the multiplicand by each digit of the multiplier and then adding up all the properly shifted results. This method requires a multiplication table for single digits available to the algorithm. Knuth's classical multiplication algorithm [15] can be stated as shown in Algorithm 2.

The complexity of the classical multiplication algorithm is $O(n^2)$. Therefore, the number representation that has fewer digits theoretically should run faster than the number representation that has more digits in its representation.

In addition, the density of nonzero digits in the numbers influences the number of addition that has to be carried out by the classical multiplication algorithm as well.

Algorithm 3 shows the modified version of Algorithm 2 that computes the squaring operation efficiently for binary numbers. The efficiency of the modified squaring algorithm comes from Steps 2.1 to 2.2.1. Since the products of $a_j \cdot a_i$ and $a_i \cdot a_j$ are the same, this product is therefore calculated just once in Step 2.2.1. Note that $(uv)_r$ in Step 2.2.1 of Algorithm 3 is the result of the addition. In Algorithm 3, v is a single-precision digit, while u is a multiple-precision digit. With this improvement, the number of partial products in the squaring algorithm is $n(n-1)/2$ less than what was found in Algorithm 1.

2.2. Karatsuba Multiplication and Squaring Algorithms. Karatsuba's algorithm is an efficient scheme for multiplying two large numbers or two polynomials. It was introduced by Karatsuba and Ofman in 1960 and published in 1962 [20]. This algorithm is a remarkable example of the divide and conquer paradigm [21, 22], specifically for its binary splitting [23]. This method requires three multiplications and four additions in each iteration. To apply the algorithm both numbers are split into a lower and an upper half (for simplicity, assume n is even):

$$\begin{aligned} A &= A_L \times r^{n/2} + A_R, \\ B &= B_L \times r^{n/2} + B_R. \end{aligned} \tag{1}$$

The halves A_R, A_L, B_R, and B_L are split again in half in the next iteration step. Since every step exactly halves the number of coefficients, the algorithm terminates after $t = \log_2 n$ steps. Algorithm 4 shows the recursive Karatsuba algorithm (assuming the lengths of A and B are even). We can use Karatsuba algorithm for squaring with small modification. Algorithm 5 shows these modifications in Steps 2-3.

Combining other multiplication algorithms with Karatsuba algorithm is another technique that has been used by researchers [24]. The study on squaring and multiplying large integers by Zuras has shown the 2-way, 3-way, and 4-way approaches for calculating big integer multiplication [25]. Sadiq and Ahmed [26] have extended the work further and summarized the results after splitting the long numbers into multi partitions (up to 10 partitions). More details on squaring algorithms can be found in the literature [6, 8, 27–29].

3. Big-Ones Representation and the Proposed Algorithms

In this section, the Big-one (Bo) integer representation and the proposed multiplication and squaring algorithms, which are based on this representation, are presented. Big-one representation is created based on the binary number representation. Big-one is a compact representation with low Hamming weight (HW) compared to the binary number representation.

A Big-one is the numeric value of a sequence of n consecutive binary symbol "1" with length n and is denoted by O_n. Examples of Bo's are $O_1 = 1_2 = 1$ and $O_3 = 111_2 = 7$. Consider

$$O_n = \overbrace{1 \cdots 1}^{n}{}_2 = \sum_{i=0}^{n-1} 2^i = 2^n - 1. \tag{2}$$

A set of all Bo's is called Big-ones' set and is denoted by $\widehat{O} = \{O_1, O_2, O_3, \ldots\} = \{1_2, 11_2, 111_2, \ldots\} = \{1, 3, 7, 15, \ldots\}$.

Big-Ones Number System (BONS). Let $A = (a_n \ldots a_2 a_1 a_0)_2$ be a number in radix 2, where $a_i \in \widehat{O} \cup \{0\}$. This number system is called Big-one number system and denoted as BONS. For example, $A = 11101010101111_2$ can be represented by $O_3 0 O_1 0 O_1 0 O_1 0000 O_4$ in BONS. This number system is redundant. To transform BONS into a canonic (not redundant) representation, the maximum length of Big-ones is used. The canonical version of BONS is known as CBONS. CBONS is a compressed representation of Big-one, by ignoring all the zeros and modifying the notation O_n to $O_{(n,P)}$, where P shows the position of the specified Big-one in the binary number. Specifically, P is the position of the least significant bit of the specified Big-one in the binary number. For example, we can write 1110101111000011_2 in CBONS as $O_{(3,13)} O_{(1,11)} O_{(4,6)} O_{(2,0)}$. To optimize the calculations based on CBONS, we can limit the length of maximum Big-ones to "w" ($O_{(n,P)}$such that $n \leq w$) which we identified in this paper as the *maximum length* of Big-ones [30–32].

3.1. Big-Ones Analysis. From the definition of CBONS, it is apparent that there will be at least one digit zero bounding from the left and at least another digit zero bounding from the right of each Big-one digit (except for the least and most significant bits). Consequently, to calculate the number of O_ls in any given binary number, we have to calculate the probability of "$0O_l0$" patterns appearing in the binary number. Since the probability of digit "1" and digit "0" appearing in a binary digit is $1/2$, therefore it follows that the probability of O_l appearing in a binary number is $P(O_l) = 1/2^{l+2}$. As a result, the number of O_ls in an n-bit binary number is $N(O_l) = n/2^{l+2}$. To calculate the Hamming weight of Bo's in a Big-one number system, it is enough to calculate the total number of Big-ones, $N(O_l)$, where $1 \leq l \leq n$. Consider

$$\begin{aligned} \text{HW (in CBONS)} &= \sum_{l=1}^{n} N(O_l) = \sum_{l=1}^{n} \frac{n}{2^{l+2}} \\ &= \frac{n}{4} \sum_{l=1}^{n} 2^{-l} = \frac{n}{4} \left(\sum_{l=0}^{n} 2^{-l} - 1 \right). \end{aligned} \tag{3}$$

Since

$$\sum_{n=0}^{m} 2^{-n} = 2^{-m} \left(2^{m+1} - 1\right), \tag{4}$$

Table 1: Comparison of the well-known polynomial multiplication algorithms.

Multiplication algorithm	Technique			Representation	Complexity
	Divide and conquer	Point evaluation	Interpolation		
Classical	—	—	—	Coefficient representation	$O(n^2)$
Karatsuba	√	—	—	Coefficient representation	$O(n^{1.58})$
Toom-Cook	√	√	√	Point-value representation	$O\left(n^{\log(2k-1)/\log k}\right)$
FFT	√	√	√	Point-value representation	$O(n \log n \log\log n)$

Table 2: The Hamming weight of Big-ones in CBONS for an 8-Kbit binary number.

	Big-one's length (w)											
	1	2	3	4	5	6	7	8	9	10	11	>12
(N_n)	1024	510	257	127	64	33	16	8	4	2	1	2
$\left(\sum_{k=1}^{n} N_k\right)$	1024	1534	1791	1918	1982	2015	2031	2039	2043	2045	2046	2048
HW/w	$\left(\frac{\sum_{k=1}^{n} N_k}{8192}\right) = \frac{2048}{8192} = \frac{1}{4}$											

(3) can therefore be written as

$$\begin{aligned} \text{HW (in CBONS)} &= \frac{n}{4}\left(2^{-n}\left(2^{n+1}-1\right)-1\right) \\ &= \frac{n}{4}\left(2-2^{-n}-1\right) = \frac{n}{4}\left(1-2^{-n}\right). \end{aligned} \tag{5}$$

For large enough n, the number of Big-ones (Hamming weight of CBONS) would be

$$\text{HW}_{\text{BONS(calculated)}} \cong \frac{n}{4}. \tag{6}$$

Table 2 shows the result of calculating the number of Big-one digits in an 8-Kbit binary number from 10,000 randomly generated binary numbers. As the table indicates, the experimental result does agree with the value found in (6).

Table 2 also indicates that the occurrence of Big-ones decreases as the length of Big-ones increases. The goal of the following experiment is therefore to find the optimized length for CBONS, to be used in LCBONS (limited length CBONS).

The length, identified as w, is important for applications such as multiplication and squaring. This is because the size of w will determine the size of the look-up table (LUT) that needs to be used by the respective algorithms. Table 3 indicates that the practical value for w is 5 since the Hamming weight when $w = 5$ is only slightly bigger than the optimum Hamming weight for CBONS (25.8% compared to 25%) but at the same time will produce a relatively compact LUT. Consequently, the following proposed multiplication and squaring algorithms will use LCBONS with $w = 5$.

3.2. Converting Binary Representation to Big-Ones Representation. Algorithm 6 shows how to convert a binary representation to CBONS representation. In Step 2.2.1, the flag *NewBo* is set to true if $a_i a_{i-1}$ = "10" and at the same time the position of the new Big-one is saved in "*pos*." In Step 2.3.1, while the flag *NewBo* is true, the length of current Big-one (*Length*) is increased by one in each iteration of the loop until $a_i a_{i-1}$ = "01" is found. The end of Big-one is identified by setting the flag *NewBo* to false in Step 2.3.2. Then, the length and position of the newly discovered Big-one digit are saved in c_{iL} and c_{ip}, respectively, where i is the position of new Big-one in array C.

Algorithm 7 is the modified version of Algorithm 6 after applying the maximum length of Big-one in BONS. In Step 2.3.3 of Algorithm 7, the length of the current Big-one digit is checked. If the length of the Big-one is bigger than w, then the relevant pointer will backtrack one bit and set the value a_i to 0. Step 2.4 of Algorithm 7 acts similar to Step 2.4 in Algorithm 6 which has been explained earlier.

To use Algorithms 6 and 7 efficiently in squaring and multiplication, we assume that the output of these algorithms is in the form of $(d_n, \ldots d_0)$, where $d_i = c_{iL}$. To show this point, we change the names of algorithms to *Bin2BO-L* and *Bin2LBO-L* accordingly.

3.3. Proposed Multiplication and Squaring Algorithm. Algorithm 8 is a modification of Algorithm 2, which has been designed based on the LBONS. In Step 1, by using function *Bin2LBO-L*, A is converted to A'. Output A' is a special representation of A in LCBONC representation that shows the length of Big-ones. Step 3.1 is introduced to ignore the zeros in A' and consequently will help reduce the number of operations. Another difference is related to Step 3.2.1.1 which uses the function LUT(a'_j, b'_i). This function fetches the product of two Big-ones by lengths of a'_j and b'_i from a precalculated look-up table.

The proposed squaring algorithm (see Algorithm 9) is a modified version of Algorithm 3. In Step 1, by executing the converter *Bin2LBO-L*, A is converted to A' which is a special representation of A in LCBONS representation with maximum length being employed ($w = 5$). Other differences are related to Step 3.1, which has been proposed by Knuth [15] to ignore the zeros in A'. Similar to Algorithm 8, in

Input: positive integer $A = (a_n, \ldots, a_0)_r$ having $n + 1$ base r digits.
Output: the square $A \cdot A = A^2 = (c_{2n+1}, \ldots, c_0)_r$ in base r representation.
Note: $(uv)_r$ are digits in base r, indicating the result of the addition.

(1) For i from 0 up to $2n + 1$ do: $c_i \leftarrow 0$.
(2) For i from 0 up to n do the following:
(2.1) Compute $(uv)_r = c_{2i} + a_i \cdot a_i$, set $c_{2i} \leftarrow v$, and $carry \leftarrow u$.
(2.2) For j from $i + 1$ up to n do the following:
(2.2.1) Compute $(uv)_r = c_{i+j} + 2a_j \cdot a_i + carry$, set $c_{i+j} \leftarrow v$, and $carry \leftarrow u$.
(2.3) $c_{i+n+1} \leftarrow u$.
(3) Return $(c_{2n+1}, \ldots, c_0)$.

// v is a single-precision digit and u is a multi-precision digit in base r

Algorithm 3: Multiple-precision classical squaring, SQ(A).

Table 3: Big-ones' distribution in an 8-Kbit binary number in LCBONS for different Big-one's length.

		Maximum size of Big-ones (w)				
		9	8	7	6	5
Big-ones' length	1	1026	1028	1032	1040	1057
	2	511	512	514	518	527
	3	258	258	259	261	266
	4	127	128	128	129	131
	5	64	64	64	65	133
	6	33	33	33	66	
	7	16	16	33		
	8	8	16			
	9	8				
HW		2051	2055	2063	2079	2114
(HW/w) %		25.0	25.1	25.2	25.4	25.8

Input: positive integers $A = (a_n, \ldots, a_0)_r$ and $B = (b_n, \ldots, b_0)_r$ having $n + 1$ base r digits.
Output: the product $C = A \cdot B$.
(1) If $n = 1$ then return $C = A \times B$.
(2) Split A, B into two equal parts:
$A = A_L \times r^{n/2} + A_R$, and $B = B_L \times r^{n/2} + B_R$.
(3) Compute the following:
$d_1 = \mathrm{KA}(A_L, B_L)$; $d_0 = \mathrm{KA}(A_R, B_R)$, and $d_{0,1} = \mathrm{KA}(A_R + A_L, B_R + B_L)$.
(4) Return $C = d_1 \times r^n + (d_{0,1} - d_0 - d_1) \times r^{n/2} + d_0$.

Algorithm 4: Recursive Karatsuba algorithm, $C = \mathrm{KA}(A, B)$.

Input: positive integers $A = (a_n, \ldots, a_0)_r$ having $n + 1$ base r digits.
Output: the product $C = A \cdot A = A^2$.
(1) If $n = 1$ then return $C = A \times A = A^2$.
(2) Split A into two equal parts A_L and A_R:
$A = A_L \times r^{n/2} + A_R$.
(3) Compute the following:
$d_1 = \mathrm{SQKA}(A_L)$; $d_0 = \mathrm{SQKA}(A_R)$, and $d_{0,1} = \mathrm{SQKA}(A_R + A_L)$.
(4) Return $C = d_1 \times r^n + (d_{0,1} - d_0 - d_1) \times r^{n/2} + d_0$.

Algorithm 5: Recursive Karatsuba squaring algorithm, $C = \mathrm{SQKA}(A)$.

Input: positive integers $A = (a_n, \ldots, a_0)_2$ having $n + 1$ base 2 digits.
Output: $C = (c_n, \ldots, c_0)_2$ having $n + 1$ digits in CBONS and non-zero $c_i = (c_{iL}, c_{iP})$.

(1) Set; $a_{-1} \leftarrow 0$, $a_{n+1} \leftarrow 0$.
(2) For i from 0 up to $n + 1$ do the following:
  (2.1) Set $Length \leftarrow 0$
  (2.2) If $a_i a_{i-1} = 10$ then do the following:
    (2.2.1) Set $NewBo \leftarrow True$; $pos \leftarrow i$.
  (2.3) While ($NewBo$) do the following:
    (2.3.1) Increase $Length$ and i by 1.
    (2.3.2) If $a_i a_{i-1} = 01$ then Set $NewBo \leftarrow False$.
  (2.4) If $Length > 0$ then do the following:
    (2.4.1) Set $c_{iL} \leftarrow Length$ and $c_{iP} \leftarrow (pos - 1)$.
(3) Return $(c_n, \ldots, c_0)$.

Algorithm 6: Binary to Big-one converter algorithm, $Bin2BO(A)$.

Input: positive integers $A = (a_n, \ldots, a_0)_2$ having $n + 1$ base 2 digits and positive integer w.
Output: $C = (c_n, \ldots, c_0)_2$ having $n + 1$ digits in LCBONS and non-zero $c_i = (c_{iL}, c_{iP})$.

(1) Set; $a_{-1} \leftarrow 0$, $a_{n+1} \leftarrow 0$.
(2) For i from 0 up to $n + 1$ do the following:
  (2.1) Set $Length \leftarrow 0$
  (2.2) If $a_i a_{i-1} = 10$ then do the following:
    (2.2.1) Set $NewBo \leftarrow True$; $pos \leftarrow i$.
  (2.3) While ($NewBo$) do the following:
    (2.3.1) Increase $Length$ and i by 1.
    (2.3.2) If $a_i a_{i-1} = 01$ then Set $NewBo \leftarrow False$.
    (2.3.3) If $Length = w$ then do the following:
      (2.3.3.1) Set $NewBo \leftarrow False$ and $a_{i-1} \leftarrow 0$.
      (2.3.3.2) Decrease i by 1.
  (2.4) If $Length > 0$ then do the following:
    (2.4.1) Set $c_{iL} \leftarrow Length$ and $c_{iP} \leftarrow (pos - 1)$.
(3) Return $(c_n, \ldots, c_0)$.

Algorithm 7: Binary to limited Big-one converter algorithm, $Bin2LBO(A)$.

Input: positive integers $A = (a_n, \ldots, a_0)_2$ and $B = (b_m, \ldots, b_0)_2$ having $n + 1$ base 2 digits.
Output: the product $A \cdot B = (c_{m+n+1}, \ldots, c_0)_2$ in base 2 representation.

(1) Compute $A' = Bin2LBO\text{-}L(A)$ and $B' = Bin2LBO\text{-}L(B)$. // $A' = (a'_n, \ldots, a'_0)$
(2) For i from 0 up to $m + n + 1$ do: $c_i \leftarrow 0$. // $B' = (b'_n, \ldots, b'_0)$
(3) For i from 0 up to m do the following:
  (3.1) Set $carry \leftarrow 0$
  (3.2) If $b_i \neq 0$ then do the following:
    (3.2.1) For j from 0 up to n do the following:
      (3.2.1.1) Compute $(uv)_b = c_{i+j} + \mathrm{LUT}(a'_j, b'_i) + carry$. // u is a multi-precision binary digit
      (3.2.1.2) Set $c_{i+j} \leftarrow v$ and $carry \leftarrow u$. // v is a single-precision binary digit
  (3.3) $c_{i+n+1} \leftarrow u$.
(4) Return $(c_{2n+1}, \ldots c_0)$.

Algorithm 8: Multiple-precision classical multiplication, $\mathrm{BOCM}(A, B)$.

Input: positive integer $A = (a_n, \ldots, a_0)_2$ having $n + 1$ base 2 digits.
Output: the square $A \cdot A = A^2 = (c_{2n+1}, \ldots, c_0)_2$ in base 2 representation.

(1) Compute $A' = Bin2LBO\text{-}L(A)$. // $A' = (a'_n, \ldots, a'_0)$
(2) For i from 0 up to $2n + 1$ do: $c_i \leftarrow 0$.
(3) For i from 0 up to m do the following:
  (3.1) If $a_i \neq 0$ then do the following:
    (3.1.1) Compute $(uv)_b = c_{2i} + \text{LUT}(a'_i, a'_i)$, and set $c_{2i} \leftarrow v$, $carry \leftarrow u$.
    (3.1.2) For j from $i + 1$ up to n do the following:
      (3.1.2.1) Compute $(uv)_b = c_{i+j} + 2\text{LUT}(a'_j, a'_i) + carry$. // u is a multi-precision binary
      (3.1.2.2) Set $c_{i+j} \leftarrow v$ and $carry \leftarrow u$. // digit and v is a single-precision
      // binary digit
  (3.2) $c_{i+n+1} \leftarrow u$.
(4) Return $(c_{2n+1}, \ldots, c_0)$

Algorithm 9: Multiple-precision classical squaring, BOSQ(A).

Step 3.1.2.1 the function $\text{LUT}(a'_j, a'_i)$ is used to fetch the product of two Big-ones from a precalculated look-up table.

4. Results and Discussion

To compute the Big-ones Hamming weight, 10,000 random numbers [33] were generated with different maximum lengths, $w = 2, \ldots, 10$, and different number lengths ranging from 32 bits to 8 Kbits. The results are summarized in Tables 2 and 3. According to this data, the Hamming weight for the numbers larger than 64 bits with $w = 5$ is about 25.8%. If we increase the value of w to 10, we can achieve slightly better Hamming weight value, that is, about 25%. However, to create a look-up table that can support $w = 10$, we have to use four times more memory than the case of $w = 5$. The size of LUT for the case of $w = 5$ is 50 bytes ($5 \times 5 \times 2$ bytes) for squaring and multiplication. In this paper, the result gathered is based on the case of $w = 5$.

Tables 2 and 3 indicate the execution time of the classical squaring (CLSQ) and multiplication (CM_MUL), Karatsuba squaring (KASQ) and multiplication (KA_MUL), and also the proposed squaring and multiplication algorithm against different bit lengths, which are randomly generated. The tests were conducted on a machine with an AMD Phenom (TM) 9950 Quad-Core processor, 3 GB RAM, Windows XP (Service Pack 3) OS, and Dev-C++ version 4.9.9.2 compiler.

According to Table 4 the proposed multiplication algorithm is more efficient than CM_MUL and KA_MUL algorithms for multiplication numbers ranging from 32 bits to 8 Kbits, which is the range of numbers used by the current number theory based cryptosystems. The proposed multiplication algorithm is about 2.3 times faster than CM_MUL for multiplying 32-bit numbers and about 3 times faster for multiplying 64-bit numbers. For numbers ranging from 128 bits to 8 Kbits, this ratio fluctuates between 3.3 and 3.9. Generally, the Karatsuba multiplication algorithm (KA_MUL) with algorithm complexity $O(n^{1.58})$ is slower than the proposed algorithm (with algorithm complexity $O(n^2)$) for multiplying numbers ranging from 32 bits to 8 Kbits. Table 4 shows that the proposed algorithm is about 7 times to 9.6 times faster than Karatsuba algorithm for multiplying 32-bit to 64-bit numbers. The speed-up ratio continuously declines from 9.6 to about 2.4 times faster for multiplying numbers in the range of 64-bit to 8-Kbit numbers.

According to Table 5, the proposed squaring algorithm is more efficient than CLSQ and KASQ algorithms for squaring numbers ranging from 32 bits to 8 Kbits. The proposed algorithm is about 2 times faster than CLSQ for squaring 32-bit numbers and this ratio gradually increases to 3.7 times for squaring 8-Kbit numbers. In general, the Karatsuba algorithm (KASQ) is slower than the proposed algorithm for squaring numbers between the ranges of 32 bits and 8 Kbits. Table 5 shows that the proposed algorithm is about 7.9 times to 10.4 times faster than Karatsuba algorithm for squaring 32-bit to 64-bit numbers. The speed-up ratio continuously declines from 10.4 to about 2.5 times faster for squaring numbers in the range of 64-bit to 8-Kbit numbers.

5. Conclusion

A multiplication and a squaring algorithm with a small look-up table, which are based on the classical multiplication algorithm and Big-ones' representation, are presented in this paper to speed up the squaring and multiplication calculation in public-key cryptography algorithms. The efficiency of the classical multiplication and squaring algorithm does not cover the whole range of numbers that is used by number theory based cryptosystems. In many instances, it has been reported that, at the threshold of 255 digits, the Karatsuba algorithm is performing better than the classical algorithm. In the proposed method, binary numbers are first converted to Big-one representation before being processed by the proposed multiplication or squaring algorithms. Compact representation with low Hamming weight of the Big-one representation decreases the number of submultiplication operations in the squaring and multiplication calculation. The experimental result gathered indicates that the proposed squaring and multiplication algorithm are efficient enough to substitute either the classical algorithm or Karatsuba algorithm or the hybrid of the two algorithms for squaring

Table 4: Execution time (msec) of multiplication algorithms.

Algorithm	Length of numbers (bits)								
	32	64	128	256	512	1024	2048	4096	8192
CM_MUL	0.007	0.024	0.082	0.316	1.15	4.41	18.01	63.8	285
KA_MUL	0.021	0.077	0.260	0.754	2.35	7.28	21.32	64.9	199
Proposed multiplication	0.003	0.008	0.028	0.087	0.32	1.34	4.66	19.6	84

Table 5: Execution time (msec) of squaring algorithms.

Algorithm	Length of numbers (bits)								
	32	64	128	256	512	1024	2048	4096	8192
CLSQ	0.00494	0.0162	0.056	0.221	0.812	3.09	12.61	44.2	192
KASQ	0.01917	0.0599	0.179	0.533	1.653	4.91	15.02	45.1	141
Proposed squaring	0.00247	0.0058	0.019	0.059	0.231	0.97	3.4	13.2	57

numbers. This finding should increase the performance of number theory based cryptosystems which depend heavily on the process of exponentiation (a process that depends on squaring and multiplication) of large integers in achieving the desired level of security.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

The researchers would like to thank the Universiti Sains Malaysia for supporting this research through Project Grant (1001/PKOMP/817059).

References

[1] N. Boudriga, "Security of mobile communications," in *Proceedings of the IEEE International Conference on Signal Processing and Communications (ICSPC '07)*, pp. li–lii, 2007.

[2] L. Li and L. Tao, "Security study of mobile business based on WPKI," in *Proceedings of the 8th International Conference on Mobile Business (ICMB '09)*, pp. 301–304, Dalian, China, June 2009.

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the Association for Computing Machinery*, vol. 21, no. 2, pp. 120–126, 1978.

[4] W. Diffie and M. E. Hellman, "New directions in cryptography," *Institute of Electrical and Electronics Engineers*, vol. 22, no. 6, pp. 644–654, 1976.

[5] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Advances in Cryptology*, vol. 196 of *Lecture Notes in Computer Science*, pp. 10–18, 1985.

[6] D. M. Gordon, "A survey of fast exponentiation methods," *Journal of Algorithms*, vol. 27, no. 1, pp. 129–146, 1998.

[7] N. Nedjah and L. de Macedo Mourelle, "A review of modular multiplication methods and respective hardware implementations," *Informatica*, vol. 30, no. 1, pp. 111–129, 2006.

[8] T.-J. Chang, C.-L. Wu, D.-C. Lou, and C.-Y. Chen, "A low-complexity LUT-based squaring algorithm," *Computers & Mathematics with Applications*, vol. 57, no. 9, pp. 1494–1501, 2009.

[9] H. Zhengbing, R. M. Al Shboul, and V. P. Shirochin, "An efficient architecture of 1024-bits cryptoprocessor for RSA cryptosystem based on modified Montgomery's algorithm," in *Proceedings of the 4th IEEE Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS '07)*, pp. 643–646, September 2007.

[10] J.-C. Bajard and L. Imbert, "A full RNS implementation of RSA," *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 769–774, 2004.

[11] N. S. Szabo and R. I. Tanaka, *Residue Arithmetic and Its Applications to Computer Technology*, McGraw-Hill, 1967.

[12] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[13] P. Barrett, "Implementing the Rivest, Shamir and Aldham public-key encryption algorithm on standard digital signal processor," in *Proceedings of CRYPTO'86*, Lecture Notes in Computer Science, pp. 311–323, 1986.

[14] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, NY, USA, 1997.

[15] E. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997.

[16] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Physics Doklady*, vol. 7, no. 7, pp. 595–596, 1963.

[17] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," *Soviet Mathematics*, vol. 3, pp. 714–716, 1963.

[18] S. A. Cook, *On the minimum computation time of functions [Ph.D. thesis]*, Department of Mathematics, Harvard University, May 1966.

[19] A. Schonhage and V. Strassen, "Schnelle Multiplikation großer Zahlen," *Computing in Science & Engineering*, vol. 7, pp. 139–144, 1971.

[20] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *USSR Academy of Sciences*, vol. 145, pp. 293–294, 1962.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 2000.

[22] A. V. Levitin, *Introduction to the Design and Analysis of Algorithms*, Addison Wesley, 2002.

[23] R. P. Brent, "Fast multiple-precision evaluation of elementary functions," *Journal of the Association for Computing Machinery*, vol. 23, no. 2, pp. 242–251, 1976.

[24] J. Von Zur Gathen and J. Shokrollahi, "Fast arithmetic for polynomials over F2in hardware," in *Proceedings of the IEEE Information Theory Workshop (ITW '06)*, pp. 107–111, Punta del Este, Uruguay, March 2006.

[25] D. Zuras, "More on squaring and multiplying large integers," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 899–908, 1994.

[26] M. Sadiq and J. Ahmed, "Complexity analysis of multiplication of long integers," *Asian Journal of Information Technology*, vol. 5, no. 2, 2006.

[27] W. Yang, P. Hseih, and C. Laih, "Efficient squaring of large integers," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E87-A, no. 5, pp. 1189–1192, 2004.

[28] N. Rampersad, J. Shallit, and M. W. Wang, "Avoiding large squares in infinite binary words," *Theoretical Computer Science*, vol. 339, no. 1, pp. 19–34, 2005.

[29] D. Zuras, "On squaring and multiplying large integers," in *Proceedings of the IEEE 11th Symposium on Computer Arithmetic*, pp. 260–271, July 1993.

[30] S. Jahani, *ZOT-M_K: a new algorithm for big integer multiplication [dissertation]*, Universiti Sains Malaysia, 2009.

[31] S. Jahani and A. Samsudin, "Karatsuba multiplication algorithm based on the big-digits and its application in cryptography," in *Proceedings of the 3rd International Conference on Cryptology & Computer Security (Cryptology '12)*, 2012.

[32] S. Jahani and A. Samsudin, "Karatsuba-ZOT multiplication algorithm and its application in cryptography," *Applied Mechanics and Materials*, vol. 241–244, pp. 2417–2423, 2013.

[33] M. Matsumoto and T. Nishimura, "Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.